Docket No. 42390P11201
Express Mail No.: EL651893989US

UNITED STATES PATENT APPLICATION FOR

A MULTITHREADED PROCESSOR CAPABLE OF IMPLICIT MULTITHREADED EXECUTION OF A SINGLE-THREAD PROGRAM

Inventors:

Haitham Akkary
Sebastien Hily

09896526.062801

Prepared by:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard, Seventh Floor
Los Angeles, California 90025
(310) 207-3800

A MULTITHREADED PROCESSOR CAPABLE OF IMPLICIT MULTITHREADED EXECUTION OF A SINGLE-THREAD PROGRAM

BACKGROUND OF THE INVENTION

Field of the Invention

[0001] This invention relates to multiprocessors, and more particularly to a method and apparatus for multithreaded execution of single-thread programs.

Description of the Related Art

[0002] In many processing systems today, such as personal computers (PCs), single chip multiprocessors (CMP) play an important roll in executing multithreaded programs. The threads that these processors may process and execute are independent of each other. For instance, threads may be derived from independent programs or from the same program. Some threads are compiled creating threads that do not have dependencies between themselves. In a multi-threading environment, however, some single-thread applications may be too difficult to convert explicitly into multiple threads. Also, running existing single-thread binaries on multi-threading processor does not exploit the multi-threading capability of the chip.

BRIEF DESCRIPTION OF THE DRAWINGS

[0003] The invention is illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. It should be noted that references to "an" or "one" embodiment in this disclosure are not necessarily to the same embodiment, and such references mean at least one.

[0004] **Figure 1** illustrates an embodiment of the invention.

[0005] **Figure 2** illustrates a commit processor of an embodiment of the invention.

[0006] **Figure 3** illustrates a speculative processor of an embodiment of the invention.

[0007] **Figure 4** illustrates a store-forwarding buffer of an embodiment of the invention.

[0008] **Figure 5** illustrates a load-ordering buffer of an embodiment of the invention.

[0009] **Figure 6** illustrates an embodiment of the invention having a system.

[0010] **Figure 7** illustrates a block diagram of an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[0011] The invention generally relates to an apparatus and method to multithreaded execution of single-thread programs. Referring to the figures, exemplary embodiments of the invention will now be described. The exemplary embodiments are provided to illustrate the invention and should not be construed as limiting the scope of the invention.

[0012] **Figure 1** illustrates one embodiment of the invention comprising multiprocessor 100. In one embodiment of the invention, multiprocessor 100 is a dual core single chip multiprocessor (CMP). Multiprocessor 100 further comprises commit central processing unit (CPU) 110, speculative CPU 120, register file buffer 130, trace buffer 140, load buffer 150 (also known as load ordering buffer), store buffer 160 (also known as store forwarding buffer), L1 cache 175, L2 cache 170, L0 instruction cache (I cache) 180, and L0 data cache (D cache) 190. In one embodiment of the invention L0 I cache 180 comprises two L0 I cache components. One of the L0 I cache 180 components is coupled to commit processor 110, and the other L0 I cache 180 component is coupled to speculative processor 120. In this embodiment of the invention, the two I cache components maintain duplicate information. In one embodiment of the invention, fetch requests are issued to L1 cache 175 from either of the L0 I cache 180 components. Lines fetched from L1 cache 175 are filled into L0 I cache 180 coupled to speculative processor 120 and commit processor 110. It should be noted that embodiments of the invention may contain any combination of cache memory hierarchy without diverging from the scope of the invention.

[0013] In one embodiment of the invention, L0 D cache 190 comprises two L0 D cache components. One of the L0 D cache 190 components is coupled to commit processor 110, and the other L0 D cache 190 component is coupled to speculative processor 120. In this embodiment of the invention, the two L0 D cache components maintain duplicate information. In this embodiment of the invention, store instructions/commands (stores) associated with speculative processor 120 are not written into L0 D cache 190. In this embodiment of the invention line read and write requests are issued to L1 cache 175 from either L0 D cache component. Lines fetched from L1 cache 175 are filled into L0 D cache 190 components coupled to commit processor 110 and speculative processor 120. Stores issued from commit processor 110 are written into the L0 D cache component coupled to speculative processor 120. By having exact copies of data in each L0 D cache component, internal snooping is not necessary.

[0014] In one embodiment of the invention, register file buffer 130 comprises an integer register buffer and a predicate register file buffer. In one embodiment of the invention the integer register file buffer comprises a plurality of write ports, a plurality of checkpoints and at least one read port. The integer register file buffer is used to communicate register values from commit processor 110 to speculative processor 120. In one embodiment of the invention, the integer register file buffer comprises eight (8) write ports, four (4) checkpoints, and one (1) read port to access any of the checkpointed contexts. In one embodiment of the invention, the integer register file buffer has an eight (8) register wide array and sixteen (16) rows. In one embodiment of the invention, the predicate register file buffer comprises a plurality of write ports, a plurality of checkpoints and at least one read port. The predicate register file buffer is used to communicate register values from commit processor 110 to speculative processor 120.and a second level register file coupled to speculative processor 120. In one embodiment of the invention, the predicate register file buffer comprises eight (8) write ports, four (4) checkpoints, and one (1) read port to access any of the checkpointed contexts. In one embodiment of the invention, the predicate register file buffer has an eight (8) register wide array and eight (8) rows.

[0015] **Figure 2** illustrates commit CPU 110. In one embodiment of the invention, commit CPU 110 comprises decoder 211, scoreboard 214, register file 212, and execution units 213. Likewise, **Figure 3** illustrates speculative CPU 120. In one embodiment of the invention, speculative CPU 120 comprises decoder 321, scoreboard 324, register file 322, and execution units 323. L2 cache 170 and L1 cache 175 are shared by commit CPU 110 and speculative CPU 120. In one embodiment of the invention, multiprocessor 100 is capable of executing explicitly multithreaded programs. In another embodiment, multiprocessor 100 is capable of executing single-threaded applications while using a multi-thread environment without converting the single-threaded application to an explicit multiple-thread.

[0016] In one embodiment of the invention, program execution begins as a single thread on one of commit CPU 110 and speculative CPU 120. In one embodiment of the invention, commit CPU 110 fetches, decodes, executes and updates register file 212, as well as issue load instructions/commands (loads) and stores to memory as instructed by the program. As the instructions are decoded, commit CPU 110 may direct speculative CPU 120 to start executing a speculative thread at some program counter value. This program counter value may be the address of the next instruction in memory, or it may be supplied as a hint by a compiler. For example, a fork at a next instruction address may be a thread forked at a call instruction. Speculative CPU 120 continues its thread execution until a program counter in commit CPU 110 reaches the same point in the program execution for which the speculative thread program counter points. Therefore, commit CPU 110 fetches, issues and commits every instruction in the program, even when an instruction belongs to a speculative thread.

[0017] In one embodiment of the invention, the dual execution architecture of multiprocessor 100 has a benefit wherein speculative CPU 120, executing farther in the program, provides highly efficient prefetch of instructions and data. Also, speculative CPU 120 determines the direction of many branches before the control flow of commit CPU 110 reaches these branches. In one embodiment of the invention, commit CPU 110 receives information on control flow direction from speculative CPU 120, and therefore, commit CPU 110 can avoid branch prediction for many branches and the associated misprediction penalty. In one embodiment of the invention, dependent and adjacent instructions executed correctly by the speculative thread can have the results concurrently committed in one commit cycle by commit CPU 110, saving time normally required to serially execute and propagate results between dependent instructions.

[0018] In one embodiment of the invention, input register values to the speculative thread are communicated through register buffer 130. All values written into register file 212, of commit CPU 110, are also written into register file buffer 130. In one embodiment of the invention when the speculative thread is spawned, a snapshot of register file 212 is available in register file buffer 130, located between commit CPU 110 and speculative CPU 120. Initially, when a speculative thread is started, none of speculative CPU 120's registers have the input value stored in them. Input registers that are needed may be read on demand from register file buffer 130. In one embodiment of the invention, scoreboard 324 in speculative CPU 120's decode stage is used to keep track of which registers are loaded from register file buffer 130, or written by the speculative thread. Those registers are valid in register file 322. All other registers are read on demand from register file buffer 130.

[0019] In one embodiment of the invention, input memory values to the speculative thread are read from the coherent cache hierarchy, allowing the speculative thread to access memory modified by the commit thread. In one embodiment of the invention, a cache coherency scheme is used where d-cache 190 is a write through cache, and L2 cache 170 is a write back cache using a MESI (M: modified; E: exclusive; S: shared; I: invalid) cache coherency protocol. One should note, however, that other cache coherency protocols may also be used in other embodiments of the invention.

[0020] Depending on the data flow in a particular program, commit CPU 110 may produce some register or memory input values after these inputs are read by the speculative thread. In one embodiment of the invention, to relax the limitations imposed by register and memory data flow, value prediction is used to provide initial input values to the speculative thread. In one embodiment of the invention, a simple value prediction method is used having passive prediction. In this embodiment, it is assumed that register and memory input values have already been produced by commit CPU 110 at the time the speculative thread is spawned.

[0021] In one embodiment of the invention, speculative results are written into register file 322 of CPU 120 as well as trace buffer 140. In one embodiment of the invention, trace buffer 140 is a circular buffer implemented as an array with head and tail pointers. In one embodiment of the invention, the head and tail pointers have a wrap-around bit. In one embodiment of the invention, trace buffer 140 has an array with one read port and one write port. In this embodiment of the invention, each entry has enough bytes to store the results of a number of instructions at least equal in number to the issue width of commit CPU 110. In this embodiment of the invention, each entry has a bit per instruction, with a second write port used to mark mispredicted loads.

[0022] In one embodiment of the invention, trace buffer 140 has one hundred-and-twenty-eight (128) entries that can each store results for six (6) instructions. In one embodiment of the invention, trace buffer 140 has four (4) partitions to support four (4) threads. In one embodiment of the invention, trace buffer 140 accommodates sixteen (16) bytes for storing two outputs per instruction, four (4) bytes to store renamed registers, and one (1) bit to mark if an instruction is a mispredicted load. In one embodiment of the invention, the mispredicted load bit can be set by six (6) write ports from load buffer 150. In one embodiment of the invention, when a thread partition is full, speculative execution is continued to prefetch into LO I cache 180 and L0 D cache 190, but results are not written into the trace buffer.

[0023] In one embodiment of the invention commit CPU 110 has scoreboard 214 that comprises one bit per register. In this embodiment of the invention, any modification of a register by commit CPU 110 between the fork point and the join point of a speculative thread causes the register scoreboard bit to be set. As commit CPU 110 retires the speculative thread results, it continuously keeps track in scoreboard 214 of all registers that are mispredicted. In this embodiment of the invention, instructions whose source register scoreboard bits are clear are safely committed into register file 212. Such instructions, even if dependent, do not have to be executed. There are some exceptions, however, such as loads and stores. Load and store exceptions have to be issued to memory execution units 213 to service cache misses and to check for memory ordering violations. Results of branch execution are also sent from speculative CPU 120 to commit CPU 110. Branch prediction in commit CPU 110 can be bypassed for some or all of the branches executed by speculative CPU 120.

[0024] In one embodiment of the invention loads and stores associated with commit processor 110 snoop load buffer 150. In one embodiment of the invention, when an instruction is replayed or if an instruction is a mispredicted load, the instructions associated destination register bit is set in scoreboard 214. When the instruction is clean, its destination register bit is cleared in scoreboard 214. Note that an instruction is clean when its sources are clean. Scoreboard 214 is cleared when all speculative thread instructions are committed.

[0025] In one embodiment of the invention, speculative CPU 120 does not issue store instructions to memory. In this embodiment of the invention, store instructions are posted in store buffer 160 and load instructions are posted in load buffer 150. In one embodiment of the invention, store buffer 160 is a fully associative store forwarding buffer. **Figure 4** illustrates the structure of store buffer 160 in one embodiment of the invention. In store buffer 160 (illustrated in **Figure 4**) each entry 410 comprises tag portion 420, valid portion 430, data portion 440, store identification (ID) 450 and thread ID portion 460. In one embodiment of the invention data portion 440 accommodates eight (8) bytes of data. In one embodiment of the invention valid portion 430 accommodates eight (8) bits. Store ID 450 is a unique store instruction ID of the last store instruction to write into an entry 410. In one embodiment of the invention, speculative loads access store buffer 160 concurrently with L0 D cache 190 access. If the load hits a store instruction in store buffer 160, L0 D cache 190 is bypassed and a load is read from store buffer 160. In this case, store ID 450 is also read out with the data.

[0026] In one embodiment of the invention, load data can be obtained by speculative processor 120 from either store buffer 160 or L0 D cache 190 associated with speculative processor 120. In one embodiment of the invention, loads are posted into load buffer 150. In this embodiment of the invention, when a load is posted, a mispredicted load bit is set in trace buffer 140 in case of load buffer 150 overflow.

[0027] In one embodiment of the invention store buffer 160 has one hundred-and-twenty-eight (128) entries, where the entries are four (4) way set associative. In one embodiment of the invention, store buffer 160 has two (2) store and two (2) load ports. In one embodiment of the invention store buffer 160 allows a partial tag match using virtual addresses for forwarding, and a full physical tag match to validate forwarding store ID's. In one embodiment of the invention store buffer 160 stores data written in data portion 440 starting from the first byte to avoid alignment delay. In one embodiment of the invention store buffer 160 has a replacement policy that replaces the oldest store upon a store miss, otherwise it replaces a hit entry. In one embodiment of the invention thread ID 460 is an index to a partition in trace buffer 140, and has a wrap around bit. In one embodiment of the invention, a global reset of thread entries is performed by using a thread ID content addressable memory (CAM) port (not shown).

[0028] In one embodiment of the invention, speculative loads are posted in load buffer 150. In one embodiment of the invention, load buffer 150 is a set associate load buffer coupled to commit CPU 110. **Figure 5** illustrates the structure of load buffer 150. In load buffer 150 (illustrated in **Figure 5**) each entry 510 comprises a tag portion 520, an entry valid bit portion 530, load ID 540, and load thread ID 550. In one embodiment of the invention, tag portion

520 comprises a partial address tag. In another embodiment, each entry 510 additionally has a store thread ID, a store ID, and a store valid bit (not shown). The Store ID is the ID of the forwarding store instruction if the load instruction has hit the store buffer 160.

[0029] In one embodiment of the invention the store ID and/or load ID 550 is an index into an entry in trace buffer 140, which is unique per instruction. In one embodiment of the invention the store valid bit is set to zero ("0") if a load hits store buffer 160. In this embodiment of the invention, the store valid bit is set to one ("1") if the load missed store buffer 160. In one embodiment of the invention, a replayed store that has a matching store ID clears (sets to "0") the store valid bit and sets the mispredicted bit in the load entry in trace buffer 140. In one embodiment of the invention, a later store in the program that matches tag portion 520 clears (sets to "0") the store valid bit and sets the mispredicted bit in the load entry in trace buffer 140. In one embodiment of the invention, a clean (not replayed) store that matches the store ID sets the store valid bit to "1" (one). In one embodiment of the invention, upon a clean (not replayed) load not matching any tag 520, or a load matching tag 520 with the store valid bit clear (set to "0"), the pipeline is flushed, the mispredicted bit in the load entry in trace buffer 140 is set to one ("1"), and the load instruction is restarted. In one embodiment of the invention, when a load entry is retired, entry valid bit portion 530 is cleared.

[0030] In one embodiment of the invention, load buffer 150 has sixty-four (64) entries that are four (4) way set associative. In one embodiment of the invention, load buffer 150 has a policy that replaces an oldest load. In one embodiment of the invention a global reset of thread entries is performed by using a thread ID CAM port (not shown).

[0031] In one embodiment of the invention, commit CPU 110 issues all loads and stores to memory execution units 213 (address generation unit, load buffer, data cache), including loads that were correctly executed by speculative processor 120. Valid load data with potentially dependent instructions could be committed, even when a load instruction issued by commit processor 110 misses L0 D cache 190. In one embodiment of the invention, a load miss request is sent to L2 cache 170 to fill the line, but the return data is prevented from writing to register file 212. In one embodiment of the invention, every load instruction accesses load buffer 150. A load miss of load buffer 150 causes a pipeline flush and a restart of the load instruction and all instructions that follow it.

[0032] In one embodiment of the invention, stores also access load buffer 150. In one embodiment of the invention, when an address matching store that also matches store ID 540, validity bit 530 is set in an entry 510. In this embodiment of the invention, a later store that hits an entry 510 invalidates the entry 510. In this embodiment of the invention when a store invalidates an entry 510, a load ID 550 is used to index trace buffer 140 to set the miss predicted load bit. In this embodiment of the invention when a load is fetched and the mispredicted load bit in trace buffer 140 is found to be set, a register bit is set in scoreboard 214. This register scoreboard bit may also be called the load destination scoreboard bit. In this embodiment of the invention, this optimization reduces the number of flushes that occur as the result of load misses in load buffer 150. One should note that commit CPU 110 concurrently reads trace buffer 140 and LO I cache 180. In this embodiment of the invention, this concurrent read of trace buffer 140 and L0 I cache 180 enables setting a scoreboard register bit in scoreboard 214 for a mispredicted load instruction in time without having to stall the execution pipeline.

[0033] In one embodiment of the invention "replay mode" execution starts at the first instruction of a speculative thread. When a partition in trace buffer 140 is becoming empty, replay mode as well as speculative thread execution are terminated. In one embodiment of the invention, instruction issue and register rename stages are modified as follows: no register renaming since trace buffer 140 supplies names; all instructions up to the next replayed instruction, including dependent instructions are issued; clean (not replayed) instructions are issued as no-operation (NOPs) instructions; all loads and stores are issued to memory, and clean instruction results are committed from trace buffer 140 to register file 130.

[0034] **Figure 6** illustrates system having an embodiment of the invention. System 600 comprises multiprocessor 100 (see **Figure 1**), main memory 610, north bridge 620, hublink 630, and south bridge 640. Typically, the chief responsibility of north bridge 620 is the multiprocessor interface. In addition, north bridge 620 may also have controllers for an accelerated graphics port (AGP), memory 610, and hub link 630, among others. South bridge 640 is typically responsible for a hard drive controller, a universal serial bus (USB) host controller, an input/output (I/O) controller, and any integrated sound devices, amongst others. In one embodiment of the invention, multiprocessor 100 contains embodiments of the invention described above.

[0035] **Figure 7** illustrates a process for an embodiment of the invention. Process 700 begins with block 170 which, starts the execution of a program thread by a first processor, such as commit processor 110. Block 720 performs fetching of commands by the first processor. Block 730 performs decoding of commands by the first processor. Block 740 instructs a second processor, such as speculative processor 120, to begin program execution of the same thread as the first processor, but at a location further in the program stream. Block 750 begins execution of the program thread by the second processor. On block 751 the second processor fetches commands. In block 752, the second processor performs decoding.

[0036] In block 753, the second processor updates a register file. In block 754, the second processor transmits control flow information to the first processor. In block 760, the first processor updates a register file. Block 770 determines whether the first processor has reached the same point of execution as the second processor. If block 770 determines that the first processor has not yet reached the same point in the program, process 700 continues with block 780 to continue execution. If block 770 determines that the first processor has reached the same point in the execution as the second processor, block 790 determines if the program is complete. If block 790 determines that the program is complete, process 700 stops, otherwise, process 700 continues at A.

[0037] With the use of embodiments of the invention discussed above, performance can be increased when executing single-threaded applications as a result of the speculative long-range multithreaded pre-fetch and pre-execution. The embodiments of the invention can be implemented with in-order and out-of-order multithreaded processors.

[0038] The above embodiments can also be stored on a device or machine-readable medium and be read by a machine to perform instructions. The machine-readable medium includes any mechanism that provides (i.e., stores and/or transmits) information in a form readable by a machine (e.g., a computer). For example, a machine-readable medium includes read only memory (ROM); random access memory (RAM); magnetic disk storage media; optical storage media; flash memory devices; electrical, optical, acoustical or other form of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). The device or machine-readable medium may include a solid state memory device and/or a rotating magnetic or optical disk. The device or machine-readable medium may be distributed when partitions of instructions have been separated into different machines, such as across an interconnection of computers.

[0039] While certain exemplary embodiments have been described and shown in the accompanying drawings, it is to be understood that such embodiments are merely illustrative of and not restrictive on the broad invention, and that this invention not be limited to the specific constructions and arrangements shown and described, since various other modifications may occur to those ordinarily skilled in the art.